

Unifying Requirements and Code: an Example

Alexandr Naumchev, Bertrand Meyer*, Victor Rivera

Innopolis University, Software Engineering Laboratory,
Innopolis, Russia

*Also ETH Zürich

{a.naumchev,b.meyer,v.rivera}@innopolis.ru
<http://university.innopolis.ru/>

Abstract. Requirements and code, in conventional software engineering wisdom, belong to entirely different worlds. Is it possible to unify these two worlds? A unified framework could help make software easier to change and reuse. To explore the feasibility of such an approach, the case study reported here takes a classic example from the requirements engineering literature and describes it using a programming language framework to express both domain and machine properties. The paper describes the solution, discusses its benefits and limitations, and assesses its scalability.

Keywords: software engineering, requirements specifications, multirequirements, Eiffel

1 Introduction

According to the standard view in software engineering, the tasks of requirements, design and implementation require distinct techniques and produce different artifacts.

What if instead of focusing on the differences we recognized the fundamental unity of the software construction process through all its stages? The principle of seamlessness (see e.g. [1]) follows from this assumption that the commonalities are more fundamental than the differences, and that it pays to use the same set of concepts, notations and tools throughout the development, from the most general and user-oriented initial steps down to the most technical tasks.

A consequence of the seamlessness principle is that requirements are just another software artifact, susceptible to many of the same techniques as code and design. In particular, assuming a modern programming language with powerful abstraction facilities, the requirements can be written in the same notation as the program.

The notion of multirequirements [2] adds to this principle the idea of using several interleaved descriptions: natural language, graphical, and formal (Eiffel text) serving as the reference.

How realistic is the seamless multirequirements approach, what are its limits, and what benefits does it bring? To help answer this question, the present article

takes the example used in a classic paper of the requirements literature, Jackson's and Zave's zoo control system, and describes it entirely in a seamless style, including the formal constraints that form a key part of the original article.

The goal of the paper is not advocacy but experimentation. The advocacy is present in the earlier references cited above. We practice a seamless approach to software construction and consider it fruitful, but the present discussion does not attempt to establish its superiority; rather it starts from the seamlessness hypothesis - in particular, the hypothesis that a single notation, Eiffel, is applicable to requirements analysis just as much as to programming - and applies this hypothesis fully and consistently to a significant example. While we draw some conclusions, the important part is the result of the experiment as presented here, enabling readers to form their own conclusions as to the benefits and limits of the approach.

Section 2 briefly explains why it is interesting to put into question the traditional separation between software development tasks. Section 3 proposes an approach to unify software development tasks by combining the approaches described in [2] and [3]. Section 4 introduces some theoretical and technical background. Section 5 presents the approach applied to an example. Finally, Section 6 concludes and mentions future work.

1.1 Summary of Contributions

Experimentation mentioned at the end of Section 1 resulted in the following key outcomes.

- An evidence suggesting that it is possible to use Multirequirements approach [2] for describing cyber-physical systems like zoo turnstile controller. At the same time, different types of exemplar statements goes far beyond just the relational statements used in [2].
- An evidence suggesting that a real programming language notation may be even more expressive than most of the popular formal notations. Section 5.5 contains all the details.
- An example showing how object orientation helps to effectively manage complexity in specifications. The approach used in [3], where the specification is basically a linear list of statements, does not scale to the case of large systems, when the number of requirements is too big. Object orientation provides a way to relate the conceptual objects so that the resulting specification will be scattered across the classes in an intuitive way.

2 The drawbacks of too much separation of concerns

Historically, there was a reason for emphasizing the distinction between development tasks. The goal was to highlight the specific needs of requirements and design, moving away from the “code first, think later” way of building software. But as the precepts of software engineering have gained wide acceptance and

programming languages have moved from low-level machine-coding notations to descriptive formalisms with high expressive power, the reverse approach is worth exploring: instead of emphasizing the differences, show the fundamental unity of the software process.

The traditional approach is subject to five criticisms.

- i) Insufficient information. Requirements analysts do not know what details are important for developers. They are good at expressing customer needs in a form the customer is ready to sign, but they typically do not know what is implementable and what is not. [4] discusses some typical flaws of natural language requirements specifications.
- ii) Lack of communication. When developers see ambiguous or contradictory elements in the requirements, they will not always go back and ask, but will often interpret the requirement according to their own understanding, which may or may not coincide with user wishes.
- iii) Impedance mismatches [1]. The use of different formalisms at different stages requires translations and creates risks of mistakes.
- iv) Impediment to change. With different formalisms, it is difficult [1] to ensure that a change at one level is reflected at other levels.
- v) Impediment to reuse. The presence of requirements as a document specific to each project may mask the commonality between projects and make the team miss potential reuse of existing developments.

3 A seamless approach

3.1 Unifying processes

Consideration of the problems listed above leads to trying a completely different approach, which recognizes that beyond the obvious differences between tasks of software development they share fundamental needs, concepts, principles, techniques. In particular, they can be addressed through a common notation. Modern programming languages are not just coding tools to talk to a machine, but powerful tools for expressing abstract concepts and modeling complex systems. The Eiffel notation used in the present work uses object-oriented principles of classes, genericity, polymorphism and inheritance, which have proved adept at describing sophisticated systems (independently of their technical programming aspects) in a modular, flexible, reusable and evolutionary way. Thanks to the presence of Design by Contract mechanisms, it can describe not only the structure of systems but their abstract semantics.

3.2 The Hypothesis

The hypothesis explored in this paper, in light of the above analysis, is that it is possible to design a software development process that:

- i) Uses for requirements the same notation and tools as for design and implementation.

- ii) Links the resulting documents (requirements, design, code) together, ensuring a major goal of software engineering: traceability.
- iii) Makes it possible to prove, formally, the correctness of the implementation against the specification.
- iv) Supports extendibility by ensuring that small changes in the requirements will cause a proportionally small change in the design and the implementation.

3.3 How to Test the Hypothesis

The present work relies on the following scenario for testing the preceding hypothesis at least in part:

- i) Propose a candidate process.
- ii) Select examples and apply the process.
- iii) Analyze the outcome.

[2] sketches such a process, based on using object orientation for representing the relationships between the conceptual objects in the requirements document. The basic idea was to have an object-oriented code along with the natural language description of a requirement. It is also possible to represent each code fragment graphically as a BON diagram [5].

[2], however, uses as example the very notion of requirements process. In other words, it is self-referential. This confers (we hope) a certain elegance to the example, but makes it look artificial. In the present paper we take a more standard example, coming from a classic requirements paper by Jackson and Zave [3].

More precisely, the requirements from the example are represented using the model-based [6] contracts-equipped [7] object-oriented [1] notation (Eiffel).

4 Theoretical and Technical Background

4.1 Design By Contract

Work [7] gives a comprehensive description of Design By Contract. Design By Contract integrates Hoare-style assertions [8] within object-oriented programs [1] constraining the data that run time objects hold. This approach equips each class feature (member) with a predicate expression, that specify its behavior, in the form of pre- and postcondition. The postcondition has to hold whenever the precondition held and the feature finished its computation before the program execution process invokes the next feature. Design By Contract equips the class itself with an invariant predicate expression which holds in all states of the corresponding objects.

4.2 Model-Based Contracts

If classical contracts are for constraining the data that run time objects actually hold, model-based contracts are “meta” contracts for constraining the objects as mathematical entities (sets, sequences, bags, relations etc.), and an execution process does not instantiate the corresponding mathematical representations at run time as parts of the objects. Model-Based Contracts are useful when it is not possible to capture all the nuances by means of classical contracts. The PhD thesis [6] gives some examples of such situations and a comprehensive description of the concept.

4.3 AutoProof

The AutoProof [9] tool is capable of formally proving the correctness of contract-equipped object-oriented programs, both classical and model-based. AutoProof proves for every routine that the conjunction of the precondition and the class invariant before invocation ensures the conjunction of the postcondition and the class invariant after invocation. The class is verified if and only if all the class features are verified.

5 Unifying the Two Worlds: an Example

Avoiding the problems analyzed in Section 2 means unifying the worlds of requirements and code in a unified framework. This section illustrates the approach. It takes the example from the work [3] and shows how to express requirements of various types in the style of work [2] - namely, using Eiffel as a formal specification language for expressing each requirement. Originally the authors used this example to demonstrate the process of deriving specifications from requirements, and the unified approach captures all the nuances of this process.

5.1 Example Overview

The authors of [3] start with giving the overall context: “...*Our small example concerns the control of a turnstile at the entry to a zoo. The turnstile consists of a rotating barrier and a coin slot, and is fitted with an electrical interface...*” This small paragraph mostly describes the relationships between the conceptual objects. Figure 1 contains specification of the context in the style of work [2].

Translating the specification from Figure 1 back to natural language using the object-oriented semantics results in almost the same initial description: “A ZOO has a TURNSTILE turnstile; a TURNSTILE has a COINSLOT coin slot and a BARRIER barrier so that coin slot has Current TURNSTILE as turnstile and barrier has Current TURNSTILE as turnstile...” COINSLOT and BARRIER hold references to the TURNSTILE instances in order to capture the “*electrical interface*” phenomena: the word “interface” means something over which the

```

class ZOO
feature
  turnstile: TURNSTILE
end

class TURNSTILE
feature
  coinslot: COINSLOT
  barrier: BARRIER
invariant
  coinslot.turnstile = Current
  barrier.turnstile = Current
end

class COINSLOT
feature
  turnstile: TURNSTILE
invariant
  turnstile.coinslot = Current
end

class BARRIER
feature
  turnstile: TURNSTILE
invariant
  turnstile.barrier = Current
end

```

Fig. 1: Expressing the context formally

parties are able to communicate with each other; communicating means sending messages to each other, and to send message to someone in the object-oriented world is to take a reference to the object and perform a qualified call on it. So at the very least the parties should hold references to each other to be able to communicate in two directions.

5.2 The Designation Set

- **Push**(e): In event e a visitor pushes the **barrier** to its intermediate position
- **Enter**(e): In event e a visitor pushes the barrier fully home and so gains entry to the **zoo**
- **Coin**(e): In event e a valid coin is inserted into the **coin slot**
- **Lock**(e): In event e the **turnstile** receives a locking signal
- **Unlock**(e): In event e the **turnstile** receives an unlocking signal

Fig. 2: The Zoo Turnstile example designation set

After stating the problem context the authors of [3] describe the *designation set*. Each designation basically corresponds to a separate type of events observed in the problem area. The authors give the designations as a set of predicates as in Figure 2. Figure 3 is an Eiffel implementation of each designation set described in Figure 2. The implementation uses Eiffel features names as labels for the events types. The natural language descriptions from Figure 2 provide heuristics on which feature should be added to which class (Figure 2 highlights the correspondence with **bold**). Each event type has an associated history - a sequence of moments in time when the events of this particular type occurred. For example, *enters* : *MML_SEQUENCE*[*INTEGER_64*] (in Figure 3) is a sequence of moments in time expressed in milliseconds when events of type *enter* took place. *MML_SEQUENCE* is a class from the *MML* (Mathematical Modeling

```

note
  model: enters
deferred class ZOO
feature
  enter
  deferred
  ensure
    enters.but_last ~ old enters
    enters.last > old enters.last
  end
  enters: MMLSEQUENCE[INTEGER_64]
end

note
  model: locks, unlocks
deferred class TURNSTILE
feature
  lock
  deferred
  ensure
    locks.but_last ~ old locks
    locks.last > old locks.last
  end
  unlock
  deferred
  ensure
    unlocks.but_last ~ old unlocks
    unlocks.last > old unlocks.last
  end
  locks: MMLSEQUENCE[INTEGER_64]
  unlocks: MMLSEQUENCE[INTEGER_64]
end

note
  model: coins
deferred class COINSLLOT
feature
  coin
  deferred
  ensure
    coins.but_last ~ old coins
    coins.last > old coins.last
  end
  coins: MMLSEQUENCE[INTEGER_64]
end

note
  model: pushes
deferred class BARRIER
feature
  push
  deferred
  ensure
    pushes.but_last ~ old pushes
    pushes.last > old pushes.last
  end
  pushes: MMLSEQUENCE[INTEGER_64]
end

```

Fig. 3: Specifying the designation set formally

Library) and denotes mathematical sequence. *MML* contains special classes for expressing model-based contracts. Although it is possible to instantiate some simple objects from these classes (like a sequence containing one element), the instances will not be modifiable. The *model* annotation is the Eiffel mechanism to represent model-based contracts (introduced in section 4.2). For instance, expression *model : enters* in Figure 3 gives a hint that *enters* feature will be used for expressing the model-based part of the contract.

The *deferred* keyword states that the specification gives only formal definitions of the events (in terms of pre- and postconditions [8]) and does not give the corresponding operational reactions of the machine on the events. The *ensure* clause is the postcondition of the feature. It describes how the system changes after reacting on an event of the corresponding type. These specifications are intuitively plausible: an event occurrence should result in extending the corresponding history with the moment in time when the event took place, and the time of the new event should be strictly bigger than the time of the previous event, as shown, for instance, by the postcondition in feature *unlock* of Figure 3. The keyword **old** is used to indicate expressions that must be evaluated in the pre-state of the routine, and \sim makes a comparison by value.

```

deferred class ZOO
feature
  turnstile: TURNSTILE
  enters: MMLSEQUENCE[INTEGER_64]
invariant
  enters.count <= turnstile.coinslot.coins.count
end

```

Fig. 4: Entries should never exceed payments

5.3 Shared Phenomena

The authors of [3] introduce the notion of shared phenomena - that is, the phenomena visible to both the world (the environment) and the machine (the notions of the world and the machine were introduced by Jackson in [10]). In the present approach this notion is covered by using the “has a” relationships between the *ZOO* and the *TURNSTILE* classes, accompanied with the model-based contracts. Namely, since a *ZOO* has a turnstile as its feature, it can see any phenomena hosted by the turnstile: *locks*, *unlocks*, *coins*, *pushes*; since a *TURNSTILE* does not hold any references to a *ZOO*, it can not observe nor control the *enter* events modeled by *ZOO*.

5.4 Specifying the System

Work [3] introduces a set of criteria by means of which it is possible to identify whether the machine is specified or not. One of the criteria states that all requirements should be expressed in terms of shared phenomena only. Requirements refinement is the process of converting the requirements stated in terms of both shared and non-shared phenomena to the form in which they are expressed in terms of shared phenomena only. Refinement process consists of identifying some laws, which hold in the environment regardless of the machine behaviour, and constraining the machine behaviour. The resulting constraints imposed on the machine together with the laws of the environment should logically imply the requirements stated in the beginning.

The authors of [3] state that the laws of the environment are always expressed in the indicative mood, while the restrictions imposed on the machine behavior are expressed in the optative mood.

All properties of the problem derived in [3] - be they optative or indicative descriptions - can be conceptually divided into the two main categories.

Properties which hold at any moment in time: an example of such property is the *OPT1* requirement (expressed in Figure 4) saying that entries should never exceed payments (the authors of [3] use *OPT** for labeling properties expressed in an optative mood). Within the present approach this requirement can be expressed in the following way. The “something always holds” semantics fits


```

deferred class BARRIER
feature
  push
  require
    not turnstile.unlocks.is_empty
    (not turnstile.locks.is_empty) implies (turnstile.unlocks.last >
                                             turnstile.locks.last)
  deferred
  end
end

```

Fig. 5: It is impossible to use locked turnstile

```

deferred class BARRIER
feature
  turnstile: TURNSTILE
  push
  deferred
  ensure
    ((old turnstile.unlocks.last > old turnstile.locks.last) and
     (pushes.count = turnstile.coinslot.coins.count))
    implies (turnstile.locks.last > pushes.last and
             (turnstile.locks.last - pushes.last) < 760)
  end
  pushes: MMLSEQUENCE[INTEGER_64]
end

```

Fig. 6: The machine locks the turnstile timely

perfectly into the semantics of Eiffel invariant: “something holds in all states of the object”, as expressed in Figure 4.

Properties which hold depending on the type of the next event to occur: the indicative property *IND2* saying that it is impossible to push the barrier if the turnstile is locked will serve as an example (the authors of [3] use *IND** for labeling properties expressed in the indicative mood). Figure 5 depicts the corresponding specification. The initial description is divided into the two different claims: first, the turnstile should be unlocked at least once, and second, if the turnstile has ever been locked, the last unlock should have occurred later than the last lock.

Real Time Properties: the authors of [3] derive several timing constraints on the events processing. For example, the *OPT7* requirement says that the amount of time between the moment when the number of the barrier pushes becomes equal to the number of coins inserted and the moment when the machine locks the turnstile should be less than 760 milliseconds. This is basically a constraint for the reaction on the *push* event: if the next *push* event uses the last coin, the

```

deferred class ZOO
feature
  turnstile: TURNSTILE_ABSTRACT
  enter
  deferred
  end
  enters: MML_SEQUENCE[INTEGER_64]
invariant
  turnstile.coinslot.coins.count > enters.count implies
    (agent enter).precondition
end

```

Fig. 7: The turnstile let people who pay enter

machine should ensure that the turnstile is locked in a timely fashion, so that a human being will not have time to enter without paying. The 760 quantity reflects the fact that it takes at least 760 milliseconds for a human being to rotate the barrier completely and enter the Zoo.

Taking this reasoning into consideration, the present specification approach handles the timing constraint by putting it into the *push* feature postcondition (as depicted in Figure 6). The antecedent of the implication assumes the situation when before the *push* event the turnstile was locked (*oldturnstile.unlocks.last* > *oldturnstile.locks.last* expression in Figure 6), and after the event occurrence the number of barrier pushes became equal to the number of coins inserted (*pushes.count* = *turnstile.coinslot.coins.count* expression in Figure 6). The consequent reflects the requirement that, having in place the situation that the antecedent describes, there should be a *lock* event which is more late than the last *push* event (*turnstile.locks.last* > *pushes.last* expression in Figure 6), and the distance between them should be less than 760 milliseconds ((*turnstile.locks.last* - *pushes.last*) < 760 expression in Figure 6).

5.5 Specifying the “Unspecifiable”

One of the requirements mentioned in [3] was *OPT2* saying that the visitors who pay are not prevented from entering the Zoo. The authors give only informal statement of this requirement: $\forall v, m, n \bullet ((Enter\#(v, m) \wedge Coin\#(v, n) \wedge (m < n)) \Rightarrow 'The machine will not prevent another Enter event')$.

The antecedent of this implication should be read like “the number of entries is less than the number of coins inserted”. The authors of [3] do not formalize the consequent and leave it in the natural language form. The present specification approach handles this requirement using standard Eiffel mechanism called *agents* (see Figure 7).

The *agent* clause treats a feature (the *enter* feature in this particular case) as a separate object so that the feature precondition becomes one of the boolean-type features of the resulting object.

6 Conclusion

Software construction involves different activities. Typically these activities are performed separately. For instance, requirements and code, as developed nowadays, seem to belong to different worlds. The case study reported in this paper shows the feasibility of unifying requirements and code in a single framework.

This paper takes the classic Zoo Turnstile example [3] and implements it using Eiffel programming language. Eiffel is used not just to express the domain properties but also the properties of the machine [10], enabling users to combine requirements and code in a single framework. This paper does not present the complete implementation of the example due to limited space. Full implementation can be reached in the GitHub project [11].

The specification approach presented in this work is suitable not only for formalizing the statements that [3] formalizes, but also for formalizing those which are not possible to formalize with classical instruments like predicate or temporal logic (like *OPT2* requirement, see Figure 7).

The present approach is not only expressively powerful - it enables smooth transition to design and implementation. GitHub project [11] contains a continuation of the present work in the form of a complete implementation of the Zoo Turnstile example.

In order to understand the benefits of the present approach better it seems feasible to evaluate it against the hypothesis stated in Section 3.2:

- i) Unity of software development tasks: indeed, all the code fragments corresponding to different specification items merged together will bring a complete design solution available at [11] (the classes ending with “_abstract”).
- ii) Traceability between the specification and the implementation: the classes ending with “_concrete” available at [11] contain the implementation and relate to the specification classes by means of inheritance.
- iii) Provability of the classes: the AutoProof system [9] is capable of formally proving both classical and model-based contracts in Eiffel. However, it is not yet capable of proving “higher-level” agents-based contracts like the one used in Figure 7 for expressing requirement *OPT2* from the work [3]. Adding this functionality to AutoProof is one of the next work items.
- iv) Extendibility of the solution: since Eiffel artifacts used in the formalizations of the requirements items correspond to their natural language counterparts directly, it is visible right away how a change in one representation will affect the second.

Speaking about scalability of the approach, a formal representation of a requirements item specified with Eiffel is as big as the scope of the item and its natural language description are, so the overall complexity of the final document should not depend on the size of the project. Anyway, this is something to test by applying the approach to a bigger project.

6.1 Future Work

The future actions plan include:

- i) to prove formally that the specifications are consistent. In particular to ensure that the features specifications preserve the invariants of their home classes; to ensure that the invariants are self-consistent. For example it should not be possible for $P(x)$ and $\neg P(x)$ to hold at the same time.
- ii) to extend the BON notation [5] so that it will be capable of expressing model-based contracts.
- iii) to design machinery for translating model-based contract-oriented requirements to their natural language counterpart so that the result will be recognizable by a human being.
- iv) to apply the approach to a bigger project.
- v) to extend AutoProof technology [9] so that it will be able to handle agents in specifications (like in Figure 7).

It seems feasible to utilize AutoProof technology [9] for achieving goal (i). AutoProof is already capable of proving that a feature implementation preserves its specification (except specifications with agents), and it seems logical to empower it with the capabilities for working solely on the specifications level. Work [12] contains a formal proof that it is possible to achieve goal (v).

As a result of implementing the plan a powerful framework for expressing all possible views on the software under construction should emerge. The threshold of success includes the possibility to generate the specification classes (their names end with “_abstract”) available at [11] automatically, using requirements documents produced according to the present process as input.

Acknowledgment This work has been supported by the Russian Ministry of education and science with the project ”Development of new generation of cloudy technologies of storage and data control with the integrated security system and the guaranteed level of access and fault tolerance” (agreement: 14.612.21.0001, ID: RFMEFI61214X0001). Also, the authors would like to thank their colleagues Alexander Chichigin and Dr. Manuel Mazzara from the Innopolis University Software Engineering Laboratory for their invaluable feedback.

References

1. B. Meyer, *Object-oriented software construction*, vol. 2. Prentice hall New York, 1988.
2. B. Meyer, “Multirequirements,” in *Modelling and Quality in Requirements Engineering (Martin Glinz Festschrift)* (N. Seyff and A. Koziolok, eds.), MV Wissenschaft, 2013.
3. M. Jackson and P. Zave, “Deriving specifications from requirements: an example,” in *Proceedings of the 17th international conference on Software engineering*, pp. 15–24, ACM, 1995.
4. B. Meyer, “On formalism in specifications,” *IEEE software*, vol. 2, no. 1, pp. 6–26, 1985.
5. K. Waldén and J. M. Nerson, *Seamless object-oriented software architecture*. Prentice-Hall, 1995.

6. N. Polikarpova, *Specified and verified reusable components*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21939, 2014, 2014.
7. B. Meyer, *Touch of Class: learning to program well with objects and contracts*. Springer, 2009.
8. C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
9. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer, “Automatic verification of advanced object-oriented features: The autoproof approach,” in *Tools for Practical Software Verification*, pp. 133–155, Springer, 2012.
10. M. Jackson, “The world and the machine,” in *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pp. 283–283, IEEE, 1995.
11. A. Naumchev, “Jackson-zave zoo turnstile implementation..” <https://github.com/anaumche/Zoo-Turnstile-Multirequirements>, 2015.
12. D. M. Nordio, *Proofs and proof transformations for object-oriented programs*. PhD thesis, Citeseer, 2009.